**CIS667- Artificial Intelligence Semester Project Report**

**Game Of Chess (using AI)**

*Name: Rashmi Chakravarthy  
NetID: rrchakra  
GitHub Link:* [*https://github.com/Rash001/AI\_Chess\_Game*](https://github.com/Rash001/AI_Chess_Game)

**Introduction**

In this project, I developed an AI-based Chess Game with a few rule modifications from the original game. To create the AI opponent, I used the MiniMax tree search algorithm(Russell et al. 146-151), which is a popular choice for creating AI players in games such as chess. To evaluate the performance of the AI, I compared it to a random AI as a baseline. The most significant experimental result was the comparison between the random AI and the tree-based AI. The tree-based AI was able to consistently outperform the random AI (above a 90%) (refer figure 7 and 8) , demonstrating the effectiveness of the MiniMax algorithm in creating an intelligent chess-playing opponent. Overall, the use of the MiniMax algorithm and the implementation of rule modifications resulted in an AI chess game that was able to provide an exceedingly good experience to its counter Baseline AI opponents however, when competing against Human player it provided a decent fight.

**Domain Description:**

*Figure 1 - The initial representation of the 8x8 chessboard*

A picture containing text, scoreboard

Description automatically generated

*Figure 2- The initial representations of the 4 different states ( 4x4 chessboard )[ 2a, 2b, 2c, 2d]*

*Figure 2a - Black and White pieces each have two Pawns, a King, and a Queen*

Text

Description automatically generated

Text

Description automatically generated

*Figure 2b - Black and White pieces each have two rooks, a King, and a Queen*

Text

Description automatically generated

*Figure 2c - Black and White pieces each have a Pawn, a Rook, a King, and a Queen*

Text

Description automatically generated

*Figure 2d - Black pieces are two Pawns, a King, and a Queen & White pieces are two Rooks, a King, and a Queen*

* The Rules of the game ( each chess piece) are designed to be as follows:
* The game starts with Black.
* Pawns can advance one or two squares forward at a time.
* Rooks can move in a straight line either horizontally or vertically, following the rules of the original game.
* Bishops can move diagonally, following the rules of the original game.
* Kings are allowed to advance two squares at a time and can only move either horizontally or vertically, but not diagonally.
* Queens can move anywhere in any direction they want, following the rules of the original game.
* A two-dimensional array is used to represent the chessboard. The columns are represented alphabetically from A to H, and the rows are represented numerically from 1 to 8. Each position on the board is represented as a string such as "A1" or "B2" (Dirk94). There are functions within the code to convert these strings to numeric values and vice versa.
* To move a piece on the board, the user must enter the name of the piece (such as BP1 for Black Pawn 1) and the position they want to move it to (using a format like 'A1' or 'A2'). A function will then determine the possible moves for that piece based on the rules of the game. If the user's input is invalid or does not follow the rules, an "invalid" message will be displayed.
* If a checkmate occurs, the game ends immediately. If the game has lasted for 40 turns, the scores of each player are calculated and the player with the highest score is declared the winner. If both players have the same score, the game is a draw. The winning player's piece color is also displayed as output.

***Initial gameplay :***

Text

Description automatically generated

*Figure 3 - An example of Human vs Human player:*

Table

Description automatically generated

*Figure 4a - An example of Human vs Tree Based AI player*

Table

Description automatically generated

*Figure 4b - An example of Human vs Baseline AI player(Random)*

A screenshot of a computer

Description automatically generated with medium confidence

*Figure 5a - An illustration of AI vs AI player ( here, TreeBased\_AI vs TreeBased\_AI )*

Once the game has reached the turn limit or a checkmate occurs, the winner will be declared as shown in the following figure. Note that during each turn, the user must enter a key to continue or 'q' or 'Q' to quit the game.

A picture containing text, receipt

Description automatically generated

*Figure 5b - An illustration of AI vs AI player ( here, TreeBased\_AI vs TreeBased\_AI )*

**Tree AI Description**:

The minimax algorithm (Russell et al. 146-151) is a search algorithm used in decision-making and game-playing in artificial intelligence. It is commonly used in chess programs to evaluate positions and select the best move.

In the context of chess, the minimax algorithm works by constructing a tree of possible moves and positions, with the current position at the root of the tree. The tree is then searched to a fixed depth (in this case, 2) to evaluate the potential outcomes of each move.

At each non-leaf node in the tree, the minimax algorithm assigns a score to the position based on the scores of its child nodes. For a given player (either white or black), the score is the maximum of the scores of the children if it is the player's turn, or the minimum if it is the opponent's turn. This reflects the idea that the player will try to maximize their score and the opponent will try to minimize it.

Once the tree has been searched to the desired depth, the minimax algorithm selects the move that leads to the position with the best score. In chess, this would typically be the move that leads to the highest score, as the goal is to win the game. The minimax algorithm with depth 2 considers only two levels of the tree: the current position and the positions that can be reached in one move. This means that it only considers the immediate consequences of each move, without considering any further downstream effects. As a result, the minimax algorithm with depth 2 is relatively simple to implement and can be used to quickly generate a decent move in chess. However, it is not as powerful as deeper minimax algorithms that consider more levels of the tree, which can provide a more accurate evaluation of the position.

***Pseudocode for chess\_minimax:***

*FUNCTION chess\_minimax(node, nplayer):*

*IF node has no children:*

*return node.score*

*ELSE:*

*INITIALIZE candidate\_nodes as an empty list*

*FOR EACH child in node.children:*

*candidate = chess\_minimax(child, nplayer)*

*ADD candidate to candidate\_nodes*

*IF player is same as caller:*

*SET node.score to maximum of candidate\_nodes*

*ELSE:*

*SET node.score to minimum of candidate\_nodes*

*END FOR*

*return node.score*

*END FUNCTION*

The chess\_minimax function is a recursive implementation of the MiniMax algorithm (Russell et al. 146-151) for an AI chess player. It takes a game tree node and the current player as arguments and returns the score for that node based on a recursive evaluation of all possible moves and countermoves. If the node has children (possible moves), the function calls itself on each child and adds the scores to a list. If the current player is an AI player, the score for the node is set to the maximum of the scores in the list and the score for the opponent’s node is set to the minimum of the scores in the list. The function returns the score for the node, which is used by the AI player to choose its next move.

***Pseudocode for createMinimaxTree:***

*FUNCTION create\_minimax\_tree(self)*

*WHILE i < maximum\_depth*

*// new\_nodes will keep track of possible states*

*INITIALIZE new\_nodes as an empty list*

*FOR EACH node in current\_nodes*

*// Iterate possible states of the current node*

*FOR EACH ps in node.possible\_states*

*// Move current piece once to find the child*

*current.move()*

*// Append child to node.chilrdren()*

*node.children.append(child)*

*Append new\_nodes to nodes\_list*

*ADD the length of new\_nodes to node\_count*

*Increment i by 1*

*FOR EACH child in nodes\_list[0][0].children*

*IF child.score is equal to total\_score*

*Append child to candidates*

*IF the length of candidates is equal to 1*

*SET chosen\_node to candidates[0].moves*

*ELSE*

*SHUFFLE candidates*

*SET chosen\_node to candidates[0].moves*

*PRINT chosen\_node*

*Call move\_piece with chosen\_node*

*END FUNCTION*

This pseudocode takes in a node object representing a node in the minimax tree, (*Minimax algorithm in Game theory: Set 1 (introduction)*) . The create\_minimax\_tree function generates a game tree for the MiniMax algorithm in an AI chess game. It starts at the current position in the game and generates possible states by moving pieces and adding the resulting child nodes to a list. It repeats this process for a predetermined number of iterations, then chooses the next move by selecting a child node with a score equal to the total possible score (a win for the AI player). If there are multiple such nodes, it shuffles them and chooses the first one. The chosen move is then printed and passed to the move\_piece function

For the baseline AI, I used a random uniform baseline , which involved in choosing valid actions uniformly at random. Here, each valid action has an equal probability of being chosen, with no consideration for the specific context or goals of the decision-making task. This approach is not particularly effective for making good decisions and is unlikely to perform significantly better. To improve the performance of the AI system, it is definitely wise to incorporate more sophisticated decision-making strategies that consider the specific goals . This could involve using techniques such as machine learning or decision tree analysis to analyze data and make informed predictions about the best course of action.

**Tree Based AI vs Baseline (Random) AI Experiments**

I divided the states of the game into two categories based on the size of the chessboard: 8x8 and 4x4. The 8x8 board has 32 pieces, with 16 pawns (8 black and 8 white), 2 kings, 2 queens, and 2 bishops of each color, and 4 rooks of each color. The 4x4 board has a different set of pieces (as depicted in Figure-2) with 4 of each type, and their moves are determined by the possible moves or the minimax algorithm (Eppes How a computerized chess opponent "thinks" - the Minimax algorithm), which aims to achieve the most beneficial state. Each game the pieces start at different states.

Chart, histogram

Description automatically generated

*Figure 6 – Scores at the end of each game*

A histogram(*Histograms#*) is a graphical representation of data that displays the frequency of different numerical values in a dataset. It is often used to visualize the distribution of a dataset. In this particular histogram, the x-axis represents the scores for each of the games and the y-axis represents the frequency count of those scores. The data displayed in the histogram appears to follow a normal distribution, which is a type of continuous probability distribution that is symmetrical around the mean. This means that the frequencies of scores are highest in the range of 200-300, with fewer scores falling outside of this range.

Chart, histogram

Description automatically generated

*Figure 7 – Result of 100 games played by each state ( there are 5 different states)*

The graph [Figure 6] compares the number of wins for two different AI players, one using a baseline algorithm(Uniformly random ) and the other using MinMax (The MinMax player consistently performs better because it selects the most favorable moves. The graph shows that there is a clear separation between the number of wins of the Baseline and MinMax AI, with the MinMax player having more victories. There is also a smaller number of instances where both players have scored equally.

*Chart, bar chart

Description automatically generated*

*Figure 8 - Result of 100 games played by each state ( there are 5 different states) represented as a bar graph*

Additionally, the bar graph [Figure – 8]shows the results of five games played at each state, comparing the performance of the tree-based AI to the baseline AI. It illustrates that the tree-based AI consistently outperforms the baseline AI.

***Output of scores of all the winners***

In the game of chess, each type of piece on the board is assigned a numerical value, or score. The pawn has a score of 20, the rook has a score of 60, the bishop has a score of 50, the queen has a score of 100, and the king has a score of 150. To determine the winner of a game, we calculate the total scores of all of the pieces on the board ( at current state ) for both players and determine which player has the maximum score.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Conclusion**

To summarize, I created a domain for an AI chess game with different states and determined the valid actions for each chess piece in a particular state. I further implemented a user interface that allows the user to select different configurations for the players, such as human vs. human, human vs. AI, or AI vs. AI players. The user can also choose different types of AI, such as a random baseline or a minimax tree based. Alongside which I implemented a feature that allows the user to process data using the tree based AI vs. baseline automatically by specifying the number of iterations and track the scores. The results showed that the minimax algorithm was able to beat the baseline with a success rate of 98%. One of the challenges I faced was the unpredictability of the moves made by the random AI, which made it difficult to achieve accurate or efficient results.

***Bibliography***

* Russell, Stuart J., et al. “The Minimax Search Algorithm.” *Artificial Intelligence: A Modern Approach, Fourth Edition*, Fourth ed., Pearson Education, Harlow, 2021, pp. 146–151.
* Dirk94.“Dirk94/Chessai: Chess Ai Written in Python. *GitHub*, https://github.com/Dirk94/ChessAI.
* “Minimax Algorithm in Game Theory: Set 1 (Introduction).” *GeeksforGeeks*, 13 June 2022, https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/.
* “NumPy Documentation#.” *NumPy Documentation - NumPy v1.23 Manual*, https://numpy.org/doc/stable/.
* “Copy in Python (Deep Copy and Shallow Copy).” *GeeksforGeeks*, 12 Oct. 2022, https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/.
* “Histograms#.” *Histograms - Matplotlib 3.6.2 Documentation*, <https://matplotlib.org/stable/gallery/statistics/hist.html>.
* “Numpy.random.randint#.” *Numpy.random.randint - NumPy v1.23 Manual*, <https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html>
* Eppes, Marissa. “How a Computerized Chess Opponent ‘Thinks’ - the Minimax Algorithm.” *Medium*, Towards Data Science, 6 Oct. 2019, https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1.